The Software Toolworks
Walt Bilofsky, Prop.

14478 GLORIETTA DRIVE
SHERMAN OAKS, CALIFORNIA 91423

TELEPHONE
(213) 986-4885

LISP/80

by Walt Bilofsky
Release 1.0
January 21, 1981

PREFACE

LISP/80 is an interpreter for LISP, a programming language widely used in artificial intelligence experimentation. It includes more than 75 built-in functions. It offers the essential LISP data structures and functions, 16 bit integer arithmetic, list operations, recursion, string operations, file I/O, and garbage collection for automatic reuse of memory.

A simple editor and file package, written in LISP, is included. It allows editing of LISP function definitions and saving them on files.

Debugging aids include trace and optional break on errors. Provision is made for loading user-supplied machine language functions callable from LISP programs.

Two simple artificial intelligence programs, written in LISP, are included: a guessing game which learns as it goes along, and a simple version of the famous ELIZA psychiatrist program which carries on a conversation.

LISP/80 was written in order to provide computer enthusiasts with an opportunity for expanding their programming skills and understanding. Thus, LISP/80 is intended primarily to be affordable, and, within that constraint, relatively feature-rich and easy to use. LISP/80's most serious limitation is its relative slowness; machine language functions can be used to overcome this, and there are faster microcomputer LISPs available in the $200 price range.

LISP/80 is patterned after the INTERLISP dialect, which is widely used on PDP-10 and DECsystem-20 computers in the artificial intelligence community.

LISP/80 provides storage capacity of about 3600 list cells and 1200 atom name characters, and more on machines with over 48K of RAM. It comes in versions for CP/M and for the HDOS operating system (release 1.5 and later) for the H8/H89/Z89 computers, and requires at least 48K of memory.

CONTENTS

INTRODUCTION

This manual contains two major sections: a brief orientation for the LISP beginner, and a LISP/80 Reference Manual. The Reference Manual contains detailed documentation of LISP/80 for reference purposes. The orientation provides some motivation for LISP, tells what it is good for, and introduces a few important LISP concepts.

A sample LISP/80 session (Section 1) allows the beginner to make LISP "do something" before starting to learn the language. Also included are an Index to Functions and a Bibliography.

Although it is hoped that this manual will be informative enough to provide a start in learning LISP, the beginner may well find it inadequate. The primary aim of the LISP/80 project with regard to the student of LISP was to make the language available at moderate cost. The LISP Orientation manual section is included so as not to leave the beginner "high and dry". However, a comprehensive tutorial introduction to LISP, of which there are many, would run to hundreds of pages. The Bibliography section lists several of these books, and the LISP novice may find one of them helpful.

1. RUNNING LISP/80 - AN EXAMPLE

Whether you are a beginner or an experienced LISP programmer, before you settle down to wade through this manual it might be comforting to see the LISP/80 interpreter run and do something. This section provides a step by step example for that purpose.

Before doing anything else, the prudent computer scientist will make a backup copy of the LISP/80 disk and place the original, with a write protect label, in a safe, cool, dust-free, non-magnetic place. (The material on the disk is copyrighted, but you are permitted to make copies as long as they are only for your own use.)

Now mount a copy of the LISP/80 disk in drive B:. [Note: HDOS users should substitute SY1: for B: throughout this example.] (NOTE: The LISP/80 distribution disk is not bootable. If you have a one disk system, copy the files from the LISP/80 disk onto a bootable disk, and omit the characters "B:" when typing in this example.)

Type the command B:LISP. (All commands should be ended by hitting the RETURN key.) LISP/80 will load, take a few seconds to compose itself, and type the prompt "_". Since LISP generally uses upper case characters, you might want to use the CAPS LOCK key at this point if your terminal has one.

First you will evaluate a simple LISP expression. Type the expression PLUS(1 2). The LISP/80 interpreter will evaluate it and type the result.

Next you will define a simple LISP function to compute factorials. Factorial n, for positive integers n, is the product of all the numbers from 1 to n. Type the following definition:

```
DEFINE ((
(FACT (N) (COND
        ((LEQP N 0) 1)
        (T (TIMES N (FACT (DIFFERENCE N 1]
```

LISP isn't fussy about how many spaces you use when typing to it, but be sure to get the parentheses right. If you have, LISP/80 will respond with (FACT).

Now type FACT(5). The answer should be 120. If you try computing FACT of numbers a little larger than 5, you may discover some limitations of LISP/80 For instance, once the maximum LISP/80 number range of -32768 to 32767 is exceeded, computations will come out wrong. Also, the definition of FACT is recursive: that is, FACT calls itself. If you try computing FACT of a sufficiently large number, you will discover that a function can't call itself recursively indefinitely.

Next you will load and run a LISP program from the disk. Type the LISP command LOAD(B:ANIMAL). The program will load and give you instructions on how to play the animal guessing game.

To terminate the LISP/80 run and exit to the operating system, hold down the CTRL shift key and type C.

2. THE LISP/80 DISTRIBUTION DISK

The LISP/80 disk contains the following files:

LISP.COM	The LISP/80 interpreter. [On HDOS, this file is LISP.ABS.]
EDIT.LSP	The LISP/80 expression editor (LISP program).
PP.LSP	A "prettyprint" LISP function for typing LISP expressions, particularly function definitions, in readable format.
ANIMAL.LSP	Animal guessing game. To run it, do LOAD (ANIMAL). This is a simple example of an artificial intelligence program which learns as it goes along.
DOCTOR.LSP	A simple version of the ELIZA psychiatrist program. This program attempts to carry on a dialogue with a "patient". It succeeds astonishingly well considering the entire program is about 60 lines of LISP. The original ELIZA program, one of the early examples of a computer exhibiting seemingly intelligent behavior, was written at M.I.T. by Joe Weizenbaum about twenty years ago.
PATCHES.DOC	A file giving the addresses which can be patched to adjust LISP/80's storage allocation, as described in Section 4.23.

The use of EDIT and PP is described in Section 5. The example in Section 1 includes simple directions on how to run LISP and try the ANIMAL program.

Users gaining experience with LISP may want to try to understand the LISP programs on these files, and even to modify and improve them.

3. AN ORIENTATION FOR THE LISP BEGINNER

3.1. WELCOME TO LISP

Welcome to LISP. If you are a newcomer to this unique computer language, you probably purchased LISP/80 because you are interested in learning about "something different" in programming languages. LISP may or may not turn out to be a useful programming tool for you. But since LISP is totally unlike BASIC, assembly language, or, probably, any language you now know, you will learn concepts and techniques that will exercise your mind and improve your skills no matter what language you wind up programming in.

Be prepared: LISP is not an easy language to learn. This part of the manual provides a brief orientation for the beginner, and attempts to introduce some of the more important concepts. It will probably be helpful, in addition, to read or more of the introductory books on LISP listed in the Bibliography section.

What follows is an introduction only. Some of the explanations leave out details for the sake of brevity and clarity. For a complete description of LISP/80 features, use the Reference Manual section.

3.2. WHAT GOOD IS LISP?

LISP has a reputation as an "artificial intelligence" (AI) experimenters' language. That is, it is suited to writing programs which deal with problems you would ordinarily expect people to cope with: problems involving concepts, situations, objects, their properties, and groups of them.

What makes LISP good for these applications? In any programming project, the approach you take to the problem can be divided into two parts:

The data representation: how to represent the objects and structures the problem deals with; and

The algorithms: how to manipulate the data in order to solve the problem.

Most programming languages have data types like string, number, and array. This is fine for data processing tasks, like producing a balance sheet or inverting a matrix, but when trying to use such data types to represent properties and groups of objects a programmer spends more effort on "fighting the language" than on dealing with the real problem.

LISP has two data types - atoms, which are numbers or names, and lists, which are made up of atoms and other lists. Lists provide a natural representation for most of the things AI programmers want to deal with. In addition, the normal style of programming in LISP, called recursion, lends itself well to the algorithms which programmers want to use to manipulate lists.

What does this mean to a LISP programmer starting to think about how to program an AI type of problem? Relatively little thought has to go into the design of a data representation. And, if he starts with the LISP data representation, an experienced LISP programmer finds it easy to express the algorithms that might provide the desired solution.

So for certain kinds of problems, programmers using LISP need to spend very little time "fighting the language" and are able to concentrate on solving the problem.

Will LISP be any use for the tasks you want to use your computer for? If you're trying to write a program to balance your checkbook, probably not. But for many interesting problems, LISP may be just right. The only way to find out is to learn LISP. And even if you don't wind up using it a lot, you will have learned techniques for writing programs and structuring data that can be used in BASIC, assembler, and other programming languages.

3.3. ATOMS

The atom is the basic unit of data in LISP. An atom is any string of letters, digits, and hyphens. (Lower case letters are allowed, but LISP doesn't generally use them, and if your terminal has a CAPS LOCK it is wise to use it when running LISP.) Some examples of atoms are:

```
A
GAMMA
AVERYLONGATOMWITH29CHARACTERS
-327
```

If an atom can be interpreted as an integer number, it is a numeric atom (similar to a numeric constant in other languages). -327 is the only numeric atom in the examples above. All other atoms are called literal atoms.

3.4. ATOMS, NAMES, AND VALUES

Literal atoms can be used as either variables or string constants. You probably know from other programming languages that a variable is a name to which a value may be assigned, and a string constant is a string of characters that you can print and do other things with.

In LISP, atoms are used as variables, and atom names serve as string constants. To see both uses, run the LISP/80 interpreter. (See Section 1 if you need instructions on how to do this.) When the prompt "_" appears, type each LISP expression shown in the following table, and try to understand what each one does. (The first one will cause an error message; that's OK.) To provide some idea of what is going on, the BASIC equivalent for each LISP expression is also shown here.

LISP Expression	Equivalent BASIC command
A1	PRINT A1
(QUOTE A1)	PRINT "A1"
(SETQ A1 (QUOTE HI-THERE))	A1 = "HI-THERE"
A1	PRINT A1

When you type A1 by itself, LISP evaluates the atom A1 and prints the value of the atom (which can be set by the SETQ function), or gives an error if the atom has not yet been given a value. When you type (QUOTE A1) or (QUOTE HI-THERE), the atom is used as a string constant.

QUOTE is a function which prevents evaluation, so the atom continues to be itself instead standing for its value. Printing an atom actually prints the name of the atom. Printing the value of the expression (QUOTE A1), for example, printed the atom A1, so its name, "A1", came out on the terminal.

Comparing the LISP commands with the BASIC equivalents may make what is going on a little clearer. There is one difference, which is that the BASIC interpreter executes commands, while LISP/80 reads expressions, evaluates them, and prints their

values. We will simply mention that difference here, and talk more about the interpreter later on.

Notice that you didn't have to say PRINT to the LISP interpreter, because it prints the values anyway. Try typing (PRINT (QUOTE A1)) and try to figure out why LISP/80 does what it does. Hint: the value of the PRINT function is the value of the expression which is given to it to print.

3.5. LISTS

A LISP list can be a simple list of atoms, like (A B -27). A list can also contain other lists: (ALPHA (X Y Z) (BETA GAMMA)). As you can see, lists are enclosed in parentheses, and atoms in a list are separated by one or more spaces.

Some of the expressions you typed to LISP in the previous section were lists.

3.6. EXPRESSIONS AND LISP FUNCTIONAL NOTATION

An expression is something that can be evaluated. You have already seen LISP's two kinds of expressions: atoms and lists. You typed expressions to the interpreter, which evaluated them and printed the values.

To evaluate an atom, the interpreter simply finds the value the atom was set to. Evaluating a list is more complicated. A list is evaluated as a function call - that is, the application of a function to zero or more arguments. Here are several examples of statements written in the BASIC language, and their equivalent LISP function call expressions. You may try typing these expressions to LISP/80 and see what happens.

BASIC	LISP
PRINT 2 + 3	(PRINT (PLUS 2 3))
LET X = 1	(SETQ X 1)
IF X = 1 PRINT "YES"	(COND ((EQ X 1) (PRINT (QUOTE YES))))
OLD "ANIMAL"	(LOAD (QUOTE ANIMAL))

All programs in LISP are expressions. A program is run by evaluating it as an expression.

3.7. TIME OUT

Notice that LISP keeps using one kind of data item to represent two different kinds of things. Atoms are used for variables and for string constants. Lists are used both as a kind of LISP data structure, and also as a way to write LISP functions, expressions, and, as you will see later, LISP programs.

This can be very confusing at first. However, since LISP programs are written as LISP lists, this makes it easy to write programs in LISP that construct and even run other LISP programs. This is particularly useful in artificial intelligence programming, where it is often necessary for a program to create a data structure describing how to do some task. What better description is there than a data structure which is a LISP program to do, or simulate, the task?

At this point, you should read Section 4.6, which describes an alternative way of typing expressions to the interpreter without having to use QUOTE as much. From now on, we will mostly use the alternative format.

3.8. FUNCTIONS OF LISTS

How can you manipulate LISP lists? Since all LISP programming is done with functions, LISP/80 contains built-in functions to perform list operations. The essential functions for list manipulation are:

NIL is an atom which is defined as the empty list, or the list without any elements. NIL may also be written (). To prove this, type (QUOTE ()) to the interpreter. (NIL always has a value: the value of NIL is NIL.)

(CONS X L) CONS is a list CONStruction function. If X is any atom or list, and L is a list, then (CONS X L) is the list consisting of X followed by the elements in L. Try the following examples on the interpreter. Remember that in each case, the outer set of parentheses is for the interpreter; it encloses the list of two arguments to CONS. For example, in CONS (4 (3)), CONS is given two arguments, 4 and (3).

```
CONS (3 NIL)          is (3)
CONS (4 (3))          is (4 3)
CONS (A (B C D))      is (A B C D)
CONS ((A B) (C D))    is ((A B) C D)
```

In each case, CONS takes the list which is its second argument, and adds on its first argument at the front. NIL is the list with no elements, so (CONS 3 NIL) is a list with one element, the atom 3.

Notice that in the last example, (A B), which is a list itself, becomes the first element of the three-member list ((A B) C D). Also notice that we are using the EVALQUOTE notation of Section 4.6; typing CONS (A (B C D)) is equivalent to typing (CONS (QUOTE A) (QUOTE (B C D))), but is a lot easier.

(CAR L) CAR returns the first element of a list. For example, CAR ((A B C)) is A. (Remember that the outer pair of parentheses in CAR ((A B C)) is for the interpreter; this means "apply CAR to the list (A B C)."

(CDR L) CDR returns the list L, minus its first element. For example, CDR ((A B C)) is (B C).

If L is a list, then (CONS (CAR L) (CDR L)) is the same list as L. To see this, type the following expressions to the interpreter:

```
(SETQ L (QUOTE (A B C)))
(CAR L)
(CDR L)
(CONS (CAR L) (CDR L))
```

See how L was used as a variable to avoid typing (A B C) over and over. Why did we type (CAR L) instead of CAR (L)? Because in order to get the value of L, which was (A B C), L had to be evaluated. So we did not want to use the EVALQUOTE form CAR (L), since the whole point of that form is not to evaluate the arguments of the function. To see the difference, type

```
CONS ((CAR L) (CDR L))
```

and compare the result to the result of the last thing in the previous example.

3.9. DEFINING FUNCTIONS

DEFINE is a function which allows you to define your own functions. Type the following expression to the interpreter:

```
SUMSQ (3 4)
```

You can tell by the result that SUMSQ is not a known function. Now type the following expression:

```
DEFINE ((
  (SUMSQ (X Y)
       (PLUS (TIMES X X) (TIMES Y Y]
```

How you arrange this long expression or break it between lines does not matter, but be sure to get the parentheses right! The character] is shorthand; it tells LISP/80 to close all the open parentheses to the left.

Now try typing SUMSQ (3 4) again. If everything has gone correctly, you have succeeded in defining the function SUMSQ, which takes two arguments and returns the sum of their squares.

Without going into a full explanation, we will just note a few things. DEFINE takes one argument, which is a list. Each element in that list is a function definition. In this example, there is one such definition, for SUMSQ. Each function definition is itself a list, with three elements: the name of a function to be defined - in this case, SUMSQ - an argument list - (X Y) - and an expression which is the function body. DEFINE defines the function SUMSQ. Subsequently, when SUMSQ appears in an expression being evaluated, the atoms in its argument list are assigned the values of the arguments given to SUMSQ in the expression, and the function body expression is evaluated. The value of that expression is the value of the function.

When you type SUMSQ (3 4), the atom X is set to the value 3, Y is set to 4, and the function body of SUMSQ is evaluated. In this case, the function body is equivalent to

```
(PLUS (TIMES 3 3) (TIMES 4 4))
```

and the interpreter types the value of this expression, or 25.

3.10. RECURSION

When an atom is set to a value by SETQ, the atom retains the value. But when an atom is an argument in a function definition, the value it gets when the function is called is strictly temporary, and the old value is restored after the function body is evaluated. To prove this, type

```
DEFINE (( (PRINTME (X) (PRINT X)) ))
```

This defines a simple function which prints its argument. Now do

```
(SETQ X (QUOTE (HI THERE)))
X
(PRINTME (QUOTE (HELLO AGAIN))
X
```

When PRINTME was called, X took on the value (HELLO AGAIN) within the body of the function. However, the old value of X was restored when PRINTME was done.

Since function arguments have their previous values restored in this way, it is perfectly legal for a function to call itself in LISP. In fact, it is rather the right way to do things. As an example, here is a function which takes a list, and returns a new list whose elements are the original list, the CDR of the original list, the CDR of that, and so on.

```
DEFINE ((
 (LISTS (L) (COND ((NULL L) NIL)
                  (T (CONS L (LISTS (CDR L]
```

The expression which forms the function body of LISTS contains three things you have not seen before: T, NULL and COND.

T is an atom whose value is T. T is used as a truth value to represent "true"; NIL is used for "false". NULL is a predicate, or truth-valued function. The value of NULL is T if its argument is the empty list NIL, and its value is NIL otherwise.

COND is a conditional. It is explained in detail in the reference manual, but its effect in LISTS is to cause LISTS to return as value either the empty list, NIL, if the argument to LISTS is NIL, and otherwise to return the CONS of the argument L with the value of (LISTS (CDR L)).

How does the function LISTS operate, then? If its argument is NIL then it just returns NIL. If its argument is a non-empty list, it computes a value with CONS, calling itself in the process but with a shorter list for an argument. So eventually it gets down to an empty list instead of going on forever.

If you type in the above definition of LISTS and then type

```
LISTS ((A B C))
```

LISP/80 will type the value

```
((A B C) (A B) (A))
```

Rather than try to figure out, step by step, how LISTS came up with this value, let's let LISP/80 tell us by tracing the execution of LISTS. Type

```
TRACE ((LISTS))
LISTS ((A B C))
```

The TRACE function tells LISP/80 to print out the arguments each time a function is called, and the value of the function each time it returns one. In this case, the trace printout looks like this:

```
1: Calling LISTS, args = ((A B C))
2: Calling LISTS, args = ((B C))
3: Calling LISTS, args = ((C))
4: Calling LISTS, args = (NIL)
4: Returns NIL
3: Returns ((C))
2: Returns ((B C) (C))
1: Returns ((A B C) (B C) (C))
((A B C) (B C) (C))
```

The first time LISTS is called, its argument - (A B C) - is not NULL, so it tries to return (CONS L (LISTS (CDR L))). In order to do that, it must call LISTS with (CDR L), which is (B C). LISTS continues down the list, calling itself over and over,

until eventually it gets called with the empty list, NIL, and returns NIL. Then each previous call of LISTS can compute the CONS and return the value from that call.

If you want to see even more of what is going on, you can execute

```
TRACE ((CAR CDR CONS NULL COND))
```

and see absolutely everything as it happens.

A function calling itself, as LISTS does, is known as recursion. In LISP, recursion provides the programming facility which many other languages accomplish by iterative statements, such as the FOR in BASIC and the DO in FORTRAN. Iterative statements are fine for stepping down an array of subscripted variables such as you find in these more "normal" programming languages. But, as the above example shows, recursion is a natural way to operate on lists.

3.11. COUNTING PARENTHESES

You may have noticed that LISP is a language of parentheses. In fact, students sometimes claim LISP stands for "Lots of Irritating Single Parentheses!" Experienced LISPers have a method for checking that parentheses are balanced in an expression. While scanning the expression from left to right, count aloud, adding one for each "(" and subtracting one for each ")". If the final number is zero, the expression is balanced. If the count ever becomes negative, there is an error somewhere in the expression.

For example:

```
While reading:  DEFINE (((PRINTME (LAMBDA (X) (PRINT X))))

Say out loud:          123        4       5 4 5       4321
```

The final count should be zero. Since it was 1, you know there is one ")" missing.

3.12. CONCLUSION

This completes the Orientation to LISP. At this point you may continue by reading the Reference Manual section of this document. If, after this brief introduction, LISP is still a complete mystery, you may wish to consult one of the more complete references listed in the Bibliography. Either way, we wish you good success, and send you on your way toward new LISP experiences. Have patience, and always count your parentheses!

4. LISP/80 REFERENCE MANUAL

4.1. RUNNING LISP/80

The LISP/80 interpreter is run by typing the LISP command. While the interpreter is running, it accepts the normal typing conventions: DELETE erases the last character typed, and ctrl-U causes whatever has been typed on the current input line to be ignored. In addition, ctrl-C will terminate the LISP/80 program and return to system command level. (Ctrl-C is typed by holding down CTRL and typing C.)

Ctrl-B will cause an interruption of LISP/80 function execution. This will usually cause a printout of the name of the function being executed (if any), and return to the top level of the interpreter. The user may also select a mode in which ctrl-B invokes the interpreter at a lower level, within the interrupted function, allowing inspection of current variable values, printout of the current function call stack, etc (see Section 4.21).

Under CP/M, if ctrl-B fails to interrupt LISP/80, the interpreter is probably in an internal loop. The programmer can cause this, for example, by applying LAST to a list which has been looped back on itself by RPLACD. Under HDOS, ctrl-B will always work.

4.2. ATOMS

The basic unit of LISP data is the atom. An atom is a string of characters which may be at most 127 characters long. Any character is legal in an atom name, but the characters space, tab, end of line, period, (,), [,], ' and % must be quoted by preceding them with a %.

Lower case characters are legal in atoms, and are distinct from upper case characters. However, lower case is rarely used in LISP.

LISP uses atoms to represent both variables and values. A string of characters is represented as the atom with that string as its name. A string which is written in many languages as "HI THERE" would be written in LISP as the atom HI% THERE. Every atom can also have a value assigned, or bound, to it.

There are two kinds of atoms: numeric and literal. A numeric atom is composed of an optional minus sign, followed by one or more digits. Numeric atoms must fall in the range -32767 to 32767. If a numeric atom exceeds this range, no error will be given but numeric results will be incorrect. The value of a numeric atom is always the number which its name represents.

Any atom which is not a numeric atom is a literal atom. Most literal atoms do not have values initially, but may have values assigned to them in various ways, the two most common being binding of function arguments, and the SETQ function. (Individual functions, such as SETQ, are documented later on in this Reference Manual.)

Two literal atoms have predefined values in LISP: NIL and T. The value of NIL is NIL and the value of T is T. These are used to represent the logical values true, or T, and false, or NIL. NIL is also used to represent the empty list.

The value of an atom is not permanent; a new value may be assigned at any time. However, it is unwise to try to change the values of T or NIL.

Some languages require variables to be declared. This is not true of LISP. There are two ways in which an atom makes itself known to LISP. An atom is created when LISP reads it, either from the terminal or from a file. An atom may also be created by use of the PACK or PACKC functions.

4.3. S-EXPRESSIONS

S-expressions are the general LISP data structure. An s-expression is one of the following:

- An atom, or
- The expression (s1 . s2) where s1 and s2 are s-expressions.

The construct (s1 . s2) is called a dotted pair. The simplest way of creating a dotted pair is the function CONS. Some examples of s-expressions are:

```
ATOM
A-LONGER-ATOM-THAN-ONE-MIGHT-USUALLY-FIND
-37
(A . B)
(A . (B . C))
((U . V) . (255 . (Y . Z)))
```

4.4. LISTS

Most s-expressions encountered in LISP are in the form of lists. A list is one of the following:

- The atom NIL, which represents the empty list, or
- The s-expression (s . l) where s is any s-expression and l is a list.

A notational convention is used to simplify the representation of lists.

```
The list NIL                    is written ().
The list (A . NIL)              is written (A).
The list (A . (B . NIL))        is written (A B).
The list (A . (B . (C . NIL))) is written (A B C).
   ... and so on.
```

4.5. OTHER S-EXPRESSION NOTATION

Extended List Notation

In general, even when an s-expression is not a list, the expression (s . e), where s is any s-expression and e is not an atom, may be written (s e). Thus, for example,

(A . (B . (C . D))) may be written (A B C . D).

When printing s-expressions, LISP/80 uses this notation.

Superbrackets

Since LISP often gives rise to expressions containing many levels of parentheses, it is convenient to have a way to abbreviate multiple parentheses. The characters [and] are called superbrackets. The open superbracket, [, has the same effect as an open parenthesis. When a close superbracket,], appears, it matches the most recent [which has not yet been matched, even if there are unmatched open parentheses in between. If a] appears when there is no matching [, it closes all open parentheses to its left.

Examples:

```
(A (B (C (D))) E)        can be written (A [B (C (D] E)
(A (B (C (D)))           can be written (A (B (C (D]
(U (V (W (X (Y)) Z)) A)  can be written (U [V (W [X (Y] Z] A)
```

QUOTE Abbreviation

The function QUOTE is used very often in LISP programming. LISP recognizes the notation 'e, where e is any s-expression, as an abbreviation for (QUOTE e).

4.6. THE LISP/80 INTERPRETER - EVALQUOTE

LISP/80 reads commands from the terminal (or from files - see LOAD, Section 4.19) through an internal function which, for historic reasons, is referred to as EVALQUOTE. This function prints the "_" prompt on the terminal, and the user may type something for EVALQUOTE to evaluate, using one of two formats:

- o A LISP expression, which is simply evaluated, or
- o A function name followed by a list of arguments, in parentheses. When this format is used, the arguments are not evaluated before the function is applied.

Typing an atom to EVALQUOTE produces its value, or an error if the atom has no value. Typing an expression, such as (PLUS 1 2), produces the value of the expression.

But often the expressions one wants to evaluate have literal arguments which must all be quoted. For example, to demonstrate the use of the function MEMBER, one might type

```
(MEMBER (QUOTE X) (QUOTE (W X Y Z)))
```

This could be abbreviated considerably by using the equivalent form which does not evaluate arguments:

```
MEMBER (X (W X Y Z))
```

Note that this form does not actually quote the arguments, but merely refrains from evaluating them. For example, typing either (QUOTE X) or QUOTE (X) to EVALQUOTE produces X. If EVALQUOTE were really prefixing a QUOTE to each of the arguments, then the latter would produce (QUOTE X).

This can lead to confusing behavior when EVALQUOTE is used with functions which evaluate their own arguments (FEXPRS and FSUBRS - see Section 4.14). One such

function is SETQ. Typing SETQ (X (A B)) to EVALQUOTE will not work, since SETQ evaluates its second argument, and will try to apply A to B. What is really wanted is (SETQ X (QUOTE (A B))), but that is not what EVALQUOTE does. (At EVALQUOTE level, SET should be used instead of SETQ.)

4.7. FUNCTIONS

LISP programming is done by writing expressions that call functions. The user may define functions in terms of other user-defined functions and a number of built-in functions.

Functions are either LAMBDA or NLAMBDA, and spread or nospread. A LAMBDA function has its arguments evaluated before the function is applied, while an NLAMBDA receives its arguments unevaluated, and may or may not evaluate each argument before returning. A spread function expects a fixed number of arguments, while a nospread function may be called with any number of arguments. Unless otherwise specified, all functions are LAMBDA spread.

If a function is called with fewer arguments than it expects, the arguments that do not appear are taken to be NIL. If a function is called with more arguments than it expects, the additional arguments are evaluated if the function is a LAMBDA, but they are ignored by the function.

An atom which is a function name has the function definition placed on its property list (see Section 4.11). Functions which are defined by machine language subroutines have the address of the subroutine stored under the property SUBR (or, for NLAMBDAs, FSUBR). Functions defined by expressions have the expression under the property EXPR (or, for NLAMBDAs, FEXPR).

Currently, all built-in functions are SUBRs or FSUBRs, and all user-defined functions are EXPRs or FEXPRs. A future version of LISP/80 may allow the user to define machine-language SUBRs and FSUBRs.

4.8. FUNCTIONS OF S-EXPRESSIONS AND LISTS

(CONS x y). The basic function for constructing s-expressions. Constructs a list cell whose CAR is x and whose CDR is y, and returns that list cell as the function value.

(CAR l). L is a list cell (i.e., not an atom). CAR returns the first element of l. If l is a list, CAR will return the first member of the list. Applying CAR to an atom produces an error.

(CDR l). L is a list cell (i.e., not an atom). CDR returns the second element of l. If l is a list, CDR will return a list consisting of l minus its first element. Applying CDR to an atom produces an error.

Note that if the value of X is a list cell and not an atom, the following equality always holds:

X is EQUAL to (CONS (CAR X) (CDR X))

LISP/80 recognizes compound CAR/CDR function names, such as CADR and CDDADAR. (CADR X), for example, is short for (CAR (CDR X)). There is no limit (short of the maximum atom name length) on the number of As and Ds that can be used to construct

such a function name.

(Seasoned LISP programmers are recognized by their facility in pronouncing these function names. CAR is pronounced like automobile; CDR is pronounced "COULD-er". CADR is pronounced "CAD-ur", CDDADAR "COULD-ud-a-DAR", and so on. The horribly un-mnemonic names CAR and CDR are historical relics of an early LISP implementation in which two address fields in a 32-bit computer memory word were used as pointers in the list cell. The machine hardware gave us the names "Contents of Address Register" and "Contents of Decrement Register".)

(QUOTE e). NLAMBDA function. QUOTE takes one argument, and returns that argument unevaluated. E.g., (QUOTE FOO) is FOO. (QUOTE e) may also be written 'e.

(PROGN a1 ... an). NLAMBDA nospread function. Evaluates a1, a2, ..., an in sequence, and returns the value of an. PROGN is used to specify more than one computation within a single expression.

(LIST a1 ... an). Nospread function. Returns the list (a1 ... an) consisting of the values of its arguments.

(APPEND p q). P and q are assumed to be lists. APPEND returns the list consisting of the elements of p followed by the elements of q. APPEND calls CONS, and does not alter list structures. If p or q are not lists, the result may not be useful, but no error will occur. See also NCONC.

(COPY e). Returns a copy of the s-expression e. The value of COPY is EQUAL to its argument, but COPY will walk over the entire list structure of e and perform a new CONS for every list cell in e, thus producing an entirely new list structure. COPY may be used to save a copy of a list before operating on it with functions that actually alter list structure.

(REVERSE l). Returns a list consisting of the elements of the list l, in reverse order. For example, (REVERSE '(A B C)) is (C B A). REVERSE of an atom is NIL.

(SUBLIS ((u1 . v1) ... (un . vn)) e). Returns the s-expression e, with substitutions made according to the first argument. This argument consists of a list of dotted pairs (ui . vi). Every occurrence of ui in e is replaced by vi. SUBLIS checks for possible substitutions only at atoms in e, so the ui should be atoms; the vi may be any s-expression. SUBLIS creates new list structure only when necessary; if there are no substitutions the value will be EQ to e.

(LAST l). Returns the last list structure in the list l. For example, (LAST '(A B C)) is (C). If l is an atom, returns NIL.

(LENGTH l). Returns the number of elements in the list l. If l is an atom, returns 0.

Some list-like s-expressions end not in NIL, but in a dotted pair: (A B . C), or, equivalently, (A . (B . C)), for example. The built-in functions of lists test for the end of the list using the predicate ATOM, rather than NILL. Thus, LAST of the above s-expression is (B . C), LENGTH is 2, and REVERSE is (B A).

4.9. PREDICATES AND LOGICAL FUNCTIONS

(ATOM a). Returns T if a is an atom, otherwise NIL.

(LITATOM a). Returns T if a is a literal atom, and NIL otherwise. A literal atom is an atom which is not a number.

(NUMBERP a). Returns T if a is a numeric atom, and NIL otherwise.

(LISTP e). Returns T if e is a list (i.e., not an atom), and NIL otherwise. (LISTP e) is always the same as (NOT (ATOM e)).

(EQ x y). Predicate which returns T if x and y are the same pointer or atom. Two numeric atoms with the same value are always EQ in LISP/80 (although this is not necessarily true in other LISP implementations). A literal atom is always EQ to itself. Two list structures are EQ only if they arose from the same CONS operation. For example, (EQ (CONS T T) (CONS T T)) is NIL.

When a variable is given a value, that value is actually the address of the list structure which represents the value. Thus, if one variable is SETQ to another, or if a variable in a function argument list is bound to a variable which is the actual argument in a function call, the two variables will be EQ.

EQ should be used in preference to EQUAL wherever it will serve the desired purpose, since it is considerably faster.

(EQUAL x y). Predicate which returns T if x and y are the same atom or equivalent list structure. EQUAL will compare list structures down to the atomic level. For example, (EQUAL (CONS T T) (CONS T T)) is T.

(NOT e). Returns T if the value of e is NIL, otherwise returns NIL. NOT is identical to NULL, and is usually used when the argument is a predicate or truth value.

(NULL e). Returns T if the value of e is NIL, otherwise NIL. NULL is identical to NOT, and is usually used when testing whether the argument is an empty list.

(AND a1 .. an). NLAMBDA nospread function. Evaluates a1, a2, ... until one is encountered which is NIL, and returns NIL. Evaluation stops at the first argument whose value is NIL. If none of the ai evaluate to NIL, AND returns the value of the last argument, an.

(OR a1 ... an). NLAMBDA nospread function. Evaluates a1, a2, ... until one is encountered which is not NIL, and returns that value. Evaluation stops at the first non-NIL argument. If all the ai evaluate to NIL, returns NIL.

(MEMBER s l). If the s-expression s is EQUAL to any element of the list l, returns T. Otherwise, returns NIL. See also: EQUAL.

4.10. ATOMS AND VALUES

As mentioned in a preceding section, every atom may have a value. The value of a numeric atom is always the number represented by the atom name. Literal atoms do not initially have values (except for T and NIL, which evaluate to themselves), and attempting to evaluate an atom which has no value results in an error.

Atoms may receive values in three ways. The value may be set using the SET or SETQ functions described in this section. An atom may have a value bound to it temporarily within a function when it is a formal argument of the function (Section 4.14) or PROG (Section 4.16). And an atom may be given a value temporarily in the optional second argument of the EVAL function (Section 4.17).

(SET a v). A is an atom, and v is an s-expression.. SET sets the current value of a to v. If a is bound in a function or PROG, SET affects the most recent active binding; otherwise SET will change the top level value of the atom. SET returns v.

(SETQ a v). NLAMBDA function. A is an atom, and v is an s-expression. SETQ evaluates v, but not a. It sets the value of a to v. Thus, (SETQ A E) is the same as (SET 'A E). If a is bound in a function or PROG, SETQ affects the last such binding; otherwise SETQ will change the top level value of the atom. SETQ returns v.

Note carefully the difference between SET and SETQ. If the following two functions are executed:

```
(SETQ X 'Y)
(SET X 'A)
```

the first sets the value of X to Y, since SETQ does not evaluate its first argument. But the second sets the value of Y to A, since SET does evaluate both arguments before performing the assignment.

For LISP experts, it should be mentioned that LISP uses a modified deep binding scheme. Variable values are stored on a pushdown stack constructed from list cells. The top level, or global, value of a variable is kept on the atom's property list, under the property VALUE% CELL.

4.11. PROPERTY LISTS

Every literal atom has associated with it a list, called a property list, which may be used to store attributes associated with that atom. The property list is of the form

```
(property1 value1 property2 value2 ... propertyn valuen)
```

where propertyi is an atom which is the name of a property, and valuei is any s-expression. Function definitions and global values assigned to atoms are among the things which the LISP/80 interpreter stores on property lists. The programmer is free to make use of this facility as well.

(GETPROPLIST a). Returns the property list of the atom a. Gives an error if a is not a literal atom. The property list is a list of the form (p1 v1 ... pn vn), where pi is an atomic property name and vi is the value of that property.

(GETPROP a prop). Returns the value of the property prop from the property list of the atom a. Gives an error if a is not an atom. GETPROP returns NIL if the property prop does not appear on the property list of a. The way to distinguish between a property which is not there and one which has the value NIL is to do

```
(MEMBER PROP (GETPROPLIST ATM))
```

(PUTPROP atm prop val). Puts the value val on the property list of atom atm under the property prop. If atm previously had the property prop, val replaces the old value. Otherwise, the property is added.

(REMPROP atm prop). Removes the property prop from the property list of atm. Returns prop if the property was found, otherwise NIL. This function alters the list structure of the property list.

The property list functions use EQ to check for the property name. Thus, although it is possible to put a property with a non-atomic name on a property list, it will not subsequently be found or removed except by a user-defined function.

4.12. ADDRESSES, LIST STRUCTURES, AND FUNCTIONS THAT ALTER THEM

Every s-expression in LISP is represented by two words (four bytes) in memory. For numeric atoms, the first word contains an identifying bit pattern, and the second word contains the value. For literal atoms, the first word holds the address in the string storage area of the atom name, and the second word points to the atom's property list. In a list cell, the first word is the address of the CAR and the second word, the CDR.

When LISP passes around s-expressions, what it actually passes is the address of the two word representation in memory. If X currently has the value (A . B), then the value of X is an address in memory of two words, the first containing the (unique) address of the atom A, and the second the (unique) address of the atom B. (LISP insures that a literal atom with a given name always refers to the same list cell address; in other words, a literal atom is always EQ to itself.)

If (SETQ Y X) is now performed, the value of Y is set to the value of X. Y now points to the same address in memory as X does, and typing either X or Y to EVALQUOTE would print (A . B).

If X is subsequently changed, say by (SETQ X 1), this changes the address which is the value of X. Y still points to the same list cell as before, and typing Y to EVALQUOTE will print (A . B), as one would expect.

The following functions, however, actually change list structure. They can be used to achieve powerful effects, but can also create confusing results.

(RPLACA e val). Replaces the CAR of the s-expression e with the s-expression val. Returns the new value of e. This function alters existing list structure, and should be used with caution, since it can alter the value of objects which point to the expression it is changing.

(RPLACD e val). Replaces the CDR of the s-expression e with the s-expression val. Returns the new value of e. This function alters existing list structure, and should be used with caution, since it can alter the value of objects which point to the expression it is changing.

An example may help to clarify the use of RPLACA and RPLACD. The following is an illustration of an actual interchange with the LISP/80 interpreter, with comments added.

```
_(SETQ Y (SETQ X '(A . B)))
(A . B)

_X
(A . B)

_Y
(A . B)

_(RPLACA X '(C . D))
((C . D) . B)

_X
((C . D) . B)

_Y
((C . D) . B)
```

X and Y are set to point to the same list structure, (A . B).

The value of X is (A . B),

and so is the value of Y.

RPLACA is used to replace the address of A in the CAR of the value of X with the address of (C . D).

X still points to the same cell, which now contains ((C . D) . B).

Since Y points to the same cell as X, its value has been changed as well!

Functions which alter list structure can be used to create reentrant lists - that is, lists which point back to themselves. For instance, performing the functions

```
(SETQ A '(X Y Z))
(RPLACD (CDDR A) A)
```

will replace the NIL at the end of the list (X Y Z) with the address of the list itself, creating an endless loop. If these expressions are typed into EVALQUOTE, the value printed by the RPLACD will be

```
(Z X Y Z X Y Z X Y Z X Y Z X Y Z ...
```

and so on, as PRINT chases around the looped list. The printing will go on forever (or until ctrl-C is typed, or, under HDOS, ctrl-B.)

It should not be assumed that reentrant lists and other tampering with list structures are always evil. Such operations are generally more efficient than copying list structures over, and can be safely used when the list being altered is not pointed to by anything else. It is often useful to change list structures that are pointed to from several places, and to create reentrant lists, but it is necessary to know what one is doing.

Another function that alters list structures is:

(NCONC p q). P and q are assumed to be lists. NCONC creates a list consisting of the elements of p followed by the elements of q, by actually altering the list structure of p. No new list cells are created in the process, but the list p may be destroyed. NCONC returns a pointer to the new list. Note that this pointer is p, except when p is NIL. NCONC is equivalent to:

```
(LAMBDA (P Q) (COND
        ((ATOM P) Q)
        (T (RPLACD (LAST P) Q)
           P]
```

4.13. ARITHMETIC FUNCTIONS AND PREDICATES

LISP/80 provides a number of functions which operate on integer numeric atoms. The allowable range of numeric atoms is -32768 to +32767. It is the responsibility of the programmer to confine arithmetic results to that range; the result of an operation which exceeds that range will be some (not specified) number in that range, but no error message will be given.

(PLUS i j). Returns the arithmetic sum of i and j. If either of the arguments is not a numeric atom, an error occurs. Note: unlike some LISP implementations, LISP/80 does not allow more than two arguments to PLUS.

(DIFFERENCE i j). Returns the numeric difference i minus j. If either of the arguments is not a numeric atom, an error occurs.

(TIMES i j). Returns the numeric product of i and j. If either of the arguments is not a numeric atom, an error occurs. Note: unlike some LISP implementations, LISP/80 does not allow more than two arguments to TIMES.

(QUOTIENT i j). Returns the numeric integer quotient of i divided by j. If j is equal to 0, the result is undefined. If the result is not a whole number, the fractional part is discarded. If either of the arguments is not a numeric atom, an error occurs.

(REMAINDER i j). Returns the numeric remainder from i divided by j. The sign of the remainder is the same as the sign of the quotient i/j. If j is equal to 0, the result is undefined. If either of the arguments is not a numeric atom, an error occurs.

(ZEROP i). Numeric predicate. Returns T if i is EQ to 0, otherwise NIL. Gives an error if i is not numeric.

(GREATERP i j). Numeric predicate. Returns T if i is greater than j, otherwise NIL. If either of the arguments is not a numeric atom, an error occurs.

(LEQP i j) Numeric predicate. Returns T if i is less than or equal to j, otherwise NIL. If either of the arguments is not a numeric atom, an error occurs.

(LESSP i j). Numeric predicate. Returns T if i is less than j, otherwise NIL. If either of the arguments is not a numeric atom, an error occurs.

(GEQP i j). Numeric predicate. Returns T if i is greater than or equal to j, otherwise NIL. If either of the arguments is not a numeric atom, an error occurs.

4.14. FUNCTION DEFINITION AND EVALUATION

(DEFINE l). Used to define user-provided functions. DEFINE takes one argument, which is a list of defining expressions for functions. Each defining expression is either of the form (name (LAMBDA args body)) (or NLAMBDA) or else (name args body).

For example, the factorial function can be defined as follows:

```
DEFINE ((
  (FACT (LAMBDA (N) (COND
            ((ZEROP N) 1)
            (T (TIMES N (FACT
                  (DIFFERENCE N 1]
```

Alternatively, one could write (FACT (N) (COND ...]. The two forms are equivalent.

DEFINE usually causes the LAMBDA expression to be stored on the property list of the function name as the value of the EXPR property. This defines a LAMBDA function - i.e., the arguments are evaluated before being passed to the function. If NLAMBDA is used instead of LAMBDA, the definition is stored as an FEXPR and the arguments are passed unevaluated to the function.

If the argument list is an atom, rather than a list, the function is nospread - i.e., the function may be called with any number of arguments, but actually receives a single argument consisting of a list of the arguments it was called with. For example, the NLAMBDA nospread function OR could be defined by the expression:

```
DEFINE ((
  (OR (NLAMBDA L (PROG (X) (RETURN (COND
        ((NULL L) NIL)
        ((SETQ X (EVAL (CAR L))) X)
        (T (APPLY 'OR (CDR L]
```

DEFINE has no magic powers as far as function definition is concerned. The functions which manipulate property lists can be used to define functions, and to alter and remove function definitions. The LISP/80 editor changes function definitions in this way.

When a function is evaluated, the atoms in the function argument list are temporarily given the values of the arguments with which the function was called. The old values, if any, of the atoms are saved on a pushdown list. The expression comprising the body of the function is evaluated. The saved values of the atoms in the argument list are restored, and the value of the function body is returned as the value of the function.

Although LAMBDA and NLAMBDA appear to be functions themselves, they are not. They are just names which indicate to the LISP interpreter that the expression which follows is a function body.

4.15. FUNCTIONS OF FUNCTIONS

(MAPLIST l f1 f2). Applies the function f1 to the list l, the CDR of l, the CDDR of l, and so on, and returns the list of values returned by f1. If the argument f2 is specified, it is a function which is used in place of CDR to step down the list l. For example, (MAPLIST '(A B C) '(LAMBDA (X) (CONS (CAR X) (CAR X]) evaluates to ((A . A) (B . B) (C . C)).

(MAPCAR l f1 f2). Identical to MAPLIST, except applies f1 to the CAR of l, the CADR of l, and so on.

(MAPCONC l f1 f2). Identical to MAPCAR, except NCONCs together the values returned by each application of f1 to form a list, and returns that list. MAPCONC is useful when there are a variable number of elements to be inserted in the result list for each evaluation of f1. For example, if X is a list, then (MAPCONC X '(LAMBDA (Y) (AND Y (LIST Y)) will return a list of all the non-NIL elements in X.

(MAPATOMS fn). Fn is a function of one argument. MAPATOMS applies fn to each atom known to the system. Thus, (MAPATOMS 'PRINT) will print the name of every known atom. Note that MAPATOMS can not tell which atoms are no longer in use but have not been garbage collected. If it is important to consider only active atoms, a COLLECT should be done before MAPATOMS is called.

4.16. LISP PROGRAMMING CONSTRUCTS

Programming in LISP consists of writing user-defined functions. Most programming languages contain constructs which provide the programmer with conditionals and branches, and LISP is no exception. As one would expect, they are all functions.

(COND (p e ... e) ... (p e ... e)). NLAMBDA nospread function. COND provides a conditional construct for LISP programming. The arguments of COND are any number of lists (p e ... e), where p is a predicate and e ... e are expressions. COND evaluates each p in turn until one of the p returns a non-NIL value. Then COND evaluates each e following that p. The value of COND is the last e evaluated. If all of the p evaluate to NIL, the value of the COND is NIL.

(PROG vlist e1 ... en). NLAMBDA nospread function. PROG provides the LISP language with a conventional sequential programming control structure. The first argument is a list of atoms, which are the local variables of the PROG. E1 ... en are atoms, which are interpreted as statement labels, or expressions, which correspond to program statements.

PROG binds the value of each variable on vlist to NIL, and then evaluates e1, e2, and so on. (Any of the ei which are atoms are considered labels and are not evaluated.) The GO function (q.v.) is a "goto" which may be used to transfer control within a PROG. The RETURN function (q.v.) terminates PROG execution, restores the previous values of the variables in vlist, and returns a value for the PROG. If PROG execution "falls off the end" by evaluating en, the PROG returns the value NIL.

The following example is a PROG which computes the LENGTH function of a list:

```
DEFINE (((LENGTH (L)
        (PROG (U V)
              (SETQ V 0)
              (SETQ U L)
            A (COND ((ATOM U) (RETURN V)))
              (SETQ U (CDR U))
              (SETQ V (PLUS V 1))
              (GO A]
```

(GO l). NLAMBDA function. Transfers control to the label l (which does not need to be quoted) within the current PROG. Since GO never "returns", it has no value. The GO need not be physically contained within a PROG, but may be in a function called from a PROG. The label l must be defined in the most recently entered PROG which has not yet been exited, or an error occurs.

(RETURN e). Returns from the current PROG. The value of the PROG is the s-expression e. If execution is not within any PROG, an error is given. Note that the RETURN need not be physically within the body of the PROG, but can be in a function which is called from the PROG body or from some other function. RETURN always returns from the most recently entered PROG which has not yet been exited.

(SELECTQ e (e1 s11 ... s1n) ... (e2 s21 ... s2n) ... deflt). SELECTQ is the switch-case construct in the LISP programming language. It is an NLAMBDA nospread function. SELECTQ first evaluates the expression e. Next, e is compared to e1 as follows. e1 is not evaluated; it is implicitly quoted. If e1 is an atom, e is checked to see if it is EQ to e1. If e1 is a list, e is checked to see if it is EQ to any element of e1. If either of these is true, expressions s11 ... s1n are evaluated, and the value of the SELECTQ is s1n. If e is not found in e1, SELECTQ goes on to e2, and so forth. If e is not found in any of the ei, deflt is evaluated and SELECTQ returns that value.

The following expression will check to see if the value of the atom LETTER is a vowel, and will return VOWEL, CONSONANT, or Y.

```
(SELECTQ LETTER
        ((A E I O U) 'VOWEL)
        (Y 'Y)
        'CONSONANT)
```

4.17. FUNCTIONS THAT EVALUATE EXPRESSIONS

It is no accident that LISP expressions are identical in form to LISP s-expressions. One of the powerful capabilities of LISP is the ability to construct an s-expression and evaluate it as an expression.

(APPLY fn args). Returns the result of evaluating the function fn with the argument list args. APPLY is a LAMBDA, so it evaluates the argument list and the function name before applying the function.

(EVAL e). Evaluates the expression e and returns its value.

4.18. STRING MANIPULATION

String manipulation in LISP is performed by operating on atom names. To obtain a string of characters, an atom is created with that string as a name.

(UNPACK a). A is an atom. UNPACK returns a list of single character atoms which make up the name of a. A may be a numeric atom.

(PACK a1 ... an). Nospread function. A1 ... an are atoms. PACK returns the atom whose name is the catenation of the names a1 ... an. For example, (PACK 'ALPHA -1) returns ALPHA-1. PACK will create a numeric atom when the name is suitable; note that creating numeric atoms outside the range -32767 to 32767 will give strange results.

(PACKC n1 ... ni). Nospread function. N1 ... ni are numeric atoms. PACKC returns the atom whose name consists of the ASCII characters whose numeric character codes are n1 ... ni. Atom names formed with PACKC can contain control characters. For example, to write the sequence ESC, p to the terminal do (PRIN1 (PACKC 27 112)). (This sequence turns on inverse video on the H19 terminal or H89.)

(NCHARS a flg). A is an atom. NCHARS returns the number of characters in the printed name of a. For example, (NCHARS 'ALPHA) returns 5. If flg is present and non-NIL, NCHARS returns the number of characters in the PRIN1-name of a. For example, (NCHARS '%(%)) is 2, but (NCHARS '%(%) T) is 4.

(CHARACTER a). Returns the numeric value of the first ASCII character in the name of the atom a.

(CHCON a). A is an atom. CHCON returns a list of numeric atoms which are the values of the ASCII character codes which make up the name of a. For example, (CHCON 'ABC) returns (65 66 67).

4.19. INPUT/OUTPUT

(PRIN2 e ch). Prints the expression e. If ch is omitted, or NIL, e is printed on the terminal. If ch is present, it is a numeric channel number obtained from OPENW, and the expression is printed on the device or file which is open on that channel. S-expressons written to a file by PRIN2 may not read back in correctly; see PRIN1.

(PRIN1 e ch). PRIN1 is similar to PRIN2, with one exception. If an atom contains a special character (i.e., one which must be preceded by a % to be inserted in the atom name), PRIN1 prints the atom as it would be typed, with % inserted as necessary. For example, the atom %(%) would be printed as () by PRIN2, but PRIN1 will print it %(%). PRIN1 is used to output s-expressions in a form suitable for reading back in.

The name of an atom, including the % characters, is referred to as the PRIN1-name of the atom.

(PRINT e ch). Identical to PRIN1, except that PRINT terminates the output line with a newline after printing e. (PRINT e) is the same as (PROGN (PRIN1 e) (TERPRI)).

(TERPRI ch). Prints an end of line character on channel ch. Ch is usually NIL (or omitted), which sends output to the terminal. Ch may also be a channel for a file or device which has been opened for writing; see OPENW.

(POSITION ch). Returns the column number in which the next character will be printed on channel ch. Following a (TERPRI), for example, POSITION returns 0. If ch is NIL (or omitted), refers to the terminal.

(TAB n min ch). Prints a sufficient number of spaces on channel ch so that the next character will be printed in column n. At least min spaces are printed. (If min is NIL or missing, min is taken to be 1.) Thus, if the current position is to the right of column n - min, a TERPRI is performed before spacing over. Setting min to a large negative number (-100, say), removes all possibility of a TERPRI occurring. If ch is NIL (or omitted), refers to the terminal. TAB will use tabs instead of spaces wherever possible.

(OPENW fname). Fname is an atom which is the name of a file or device. OPENW attempts to open that file or device for writing, and returns a channel number if successful. S-expressions may be written to the file or device by passing the channel number to PRINT, PRIN1 or PRIN2. If no extension is specified for the filen me, the extension .LSP is assumed. In specifying an extension, remember that the character "." in an atom name must usually be quoted by preceding it with a "%". If the file or device can not be opened, an error occurs. A maximum of three files or devices may be open for reading and/or writing at any one time; to use more see CLOSE.

A file or device which has been written to must be closed before exiting from the LISP/80 interpreter, or the information written will be lost. See CLOSE. It is not necessary to close channels which have only been opened for reading, but at most three channels can be open at any one time.

(CLOSE ch). Ch is a channel obtained from a previous call to OPENW or OPENR. If ch is a number, CLOSE returns T and closes the file or device which is open on that channel. If ch is not a number, CLOSE returns NIL and does nothing.

(OPENR fname). Fname is an atom which is the name of a file or device. OPENR attempts to open that file or device for reading, and returns a channel number if successful. S-expressions may be read from the file or device by passing the channel number to READ, q.v. If the open can not be performed, an error occurs. If no extension is specified for the filename, the extension .LSP is assumed. A maximum of three files or devices may be open for reading and/or writing at any one time; to use more see CLOSE.

(READ ch). Reads an s-expression. If ch is missing or NIL, reads from the terminal. If ch is a channel number obtained from OPENR, reads from the file or device which is open on that channel. See also: READC, OPENR. When the last s-expression on the file has been read, READ returns the atom ^. Attempting to read anything else after that causes an error.

(READC ch). Similar to READ, but returns the atom whose name is the next character read from the terminal (ch missing or NIL) or the file open on channel ch.

(LOAD fname). Fname is an atom which is the name of a file. LOAD opens the file, and evaluates the s-expressions on the file as if they had been typed to the interpreter. LOAD is useful for loading programs that have been typed onto a file. LOAD returns fname.

4.20. COMMENTS

It is good programming practice to include comments in every program. LISP does not have any special way to do this, but there is a trick which can accomplish the same thing using the QUOTE function, as in the following function definition example:

```
(FACT (N) (COND ((ZEROP N)
                 (QUOTE Return 1 for (FACT 0)) 1)
                (T (QUOTE Otherwise, recurse)
                   (TIMES N (FACT (DIFFERENCE N 1]
```

There are two things to remember about this way of commenting a program. First, the comment will take up list space, and especially character storage space, that could otherwise be used for program and data. This is why the LISP program files on the LISP/80 distribution disk do not contain comments.

Second, the QUOTE function is evaluated during program execution, just like any other function. Thus, it must be used only in places where it does not affect the value of the expression in which it is inserted. This is the case in the example above.

4.21. TRACE, BREAK, ERRORS, AND PROGRAM TERMINATION

LISP/80 provides several facilities to help with debugging and examining the operation of user programs. TRACE displays call and return values of designated functions while the interpreter is running. BREAK allows examination of variable values from inside a function which is being executed. This may also be done when an error occurs.

(TRACE l). L is a list of function names. TRACE turns on tracing for each of these functions. When a traced function is called, its name and argument values are printed, along with the function call depth (counting only traced functions). When a traced function returns, the call depth and function value are printed. Both user-defined and built-in functions may be traced. TRACE returns its argument list as value.

TRACE operates by placing the property TRACE on the property list of each traced function, with property value T. In order to speed function execution, the interpreter does not test for this property unless TRACE has been called at least once. Thus, it is possible for a user function to turn tracing of individual functions on and off dynamically, but if this is done the interpreter must be signalled to look for the TRACE property by first calling (TRACE NIL).

(UNTRACE l). L is a list of function names. Turns off tracing of each named function in l (whether or not it was on). See TRACE.

(BREAK). This function, when executed, calls EVALQUOTE, which prompts the user for input exactly as at the top level of the interpreter, except that the character ":" is used as the prompt. The user may type expressions to be evaluated. Any variables which are bound at the time BREAK is called are still defined, and may be examined, and their values changed, by the user. The user may also type the following special commands:

BT — Backtrace. Types a list of all functions which are currently called, starting with the function containing the BREAK and proceeding up to the function originally called from EVALQUOTE.

CONTINUE Continue with program execution. Return from the BREAK with the value NIL.

^ Pop up to the top level (or to the next level of EVALQUOTE, if the current BREAK was caused by something typed at a previous BREAK).

Ordinarily, the effect of an error is to print the error message, abort program execution, and return to EVALQUOTE level. However, if the value of the atom BREAK is other than NIL, after an error message is printed a BREAK occurs. If the user CONTINUEs from the break, the function which caused the error returns the value NIL, and execution continues from that point.

At any time during interpreter execution, typing ctrl-B on the terminal causes function evaluation to be interrupted. (Under CP/M, if the interpreter is locked in a tight internal loop it can not be interrupted.) This interruption is handled precisely like an error. That is, if BREAK is set to NIL, ctrl-B will cause a return to top EVALQUOTE level. If BREAK has been set to other than NIL, ctrl-B will cause a BREAK and drop into EVALQUOTE. From this point, CONTINUE will resume function execution from the point of interruption.

(LOGOUT). Terminates LISP/80 interpreter execution and returns to monitor command level. All data in memory is lost.

4.22. GARBAGE COLLECTION

As the interpreter runs, new lists are created. Eventually all the available space is used up. At this point, the interpreter looks around for any list cells which were once used but are no longer needed. (This can happen, for example, if a cell was the value of an atom which was then set to a different value.) These cells are reclaimed and made available for reuse. This process is called garbage collection. (LISP was invented before the term "recycling" came into general use.)

LISP/80 divides its data storage into two areas: list cells and atom character name space. When the garbage collector runs, it prints out the amount of each kind of space it was able to make available. If a program is large or creates a lot of data, all of one kind of space can be used, and the garbage collector can not free up any at all.

It is possible to readjust the allocation between list and character space (see Section 4.23). But if both kinds of space run short, the program is really too big for the machine. At that point you should consider swapping function definitions or list structures out to the disk using READ and WRITE, or buying more memory or a PDP-10 computer.

Garbage collect is invoked automatically whenever more space is needed. It may also be run explicitly. One use of this is to print out the amount of space available.

(COLLECT). Causes a garbage collect to take place, reclaiming any atom and list cell space no longer in use. Causes the amount of space available to be printed on the terminal; see also GCGAG.

(GCGAG flg). Controls printing of garbage collection messages. Normally, a message is printed on the terminal during each garbage collection. Calling GCGAG with flg = T will suppress printing of messages. Calling GCGAG with flg = NIL will resume printing. GCGAG returns the previous value of its flag, so that a function may control the message during execution of the function and then restore the previous status on exiting.

4.23. STORAGE ALLOCATION

When the LISP interpreter is run, it divides all available storage into three areas: lists, characters, and stack. The list area holds atoms and list cells, using four bytes per item. The character area contains atom names, with an atom taking the space for its name plus three and a half bytes. The stack area is the hardware program stack, and is used only for internal subroutine linkage and storage; the LISP stack is kept in the list area. The character space and the stack space grow toward a common boundary, so that the maximum recursion depth is increased when character storage is relatively uncluttered, and may occasionally be reduced before a garbage collect.

When the interpreter exhausts list or character space, the garbage collector will show this by first displaying small or zero amounts of free or character cells, and then by giving an error message. When the program stack is exhausted, a "Stack Overflow" message appears.

Typically, LISP/80 running on a 48K system will have available about 3600 list cells, 1200 character bytes, and a minimum of 1500 bytes for the internal stack. Since a particular application may require a different allocation of available memory, these parameters may be adjusted by the user.

A permanent change can be effected by patching the file LISP.COM [under HDOS, LISP.ABS]. Type file PATCHES.DOC on the LISP/80 distribution disk to see the addresses to patch in your version and the default values, and for instructions on how to patch program files on your operating system.

There are two values which can be patched. One holds the number of list cells to be allocated. If this number is 0 (the default), there will be about two list cells allocated for every character byte. If this number is patched to a nonzero value, that number of list cells is allocated. Because of the initialization process, the actual number of list cells available (as shown by a COLLECT() upon starting LISP) will differ slightly from the number requested.

The other value holds the number of bytes assigned to the program stack. This number is initially set to 1500. All free memory not used for stack or lists is used for character storage.

The size of these areas may also be determined at the time LISP is run. If LISP is invoked by the command

```
LISP L=nnnn S=mmmm
```

where nnnn and mmmm are decimal numbers, then nnnn list cells and mmmm bytes of stack space are reserved. If the default value is acceptable, either or both of the L=nnnn and the S=mmmm may be omitted.

4.24. WRITING ASSEMBLY LANGUAGE SUBRS

User-coded machine language routines may be loaded at the time LISP/80 is run, and called as SUBRs or FSUBRs from LISP functions. This section describes how to accomplish this, assuming the reader is an accomplished machine language programmer.

To write such routines, it is necessary to understand a little about the internals of LISP/80. A list cell is two consecutive words, always starting on an address whose two low bits are 0. The first word holds the address of the CAR of the cell, and the second holds the CDR.

An atom is a list cell with the low bit of the first word set to 1. This distinguishes it from a normal list cell. In a numeric atom, the first word contains 1 and the second word holds the value. In a literal atom, the first word (with the low bit masked out) points to the atom name, and the second word to the property list. The atom name begins on an even address, and is stored as the address of the atom cell, followed by the name itself, terminated by one or two zero bytes.

LISP/80 is written in C/80, and uses that language's subroutine calling conventions. The calling sequence is: PUSH arg1; ...; PUSH arg$_n$; CALL subr; POP; ...; POP. Subroutines return their value in HL. No registers are preserved through subroutine calls. The arguments to and values returned by LISP machine language functions should be the addresses of list cells.

To write machine language functions it is necessary to know certain internal addresses. Running LISP using the command "LISP P" will print these addresses. ORG is the origin for user functions; NIL is the address of a word of memory containing the address of the atom NIL. (The address of the atom itself may change from run to run.)

The other addresses are internal routines which may be useful. Their arguments may be list cells or not, as noted.

ROUTINE	#ARGS	FUNCTION
getatom	1	Argument is address of 0-terminated atom name; returns pointer to the atom in HL.
getcell	0	Returns address of a fresh list cell in HL.
box	1	Argument is number; returns numeric atom with that value.
push	1	Argument is a list cell; pushes it on list stack.
pop	0	Pops top item of list stack into HL.

Push and pop are useful in protecting temporary list structures from a garbage collect. A collect can happen any time storage is used: in a call to getatom, getcell, box, or any LISP function that calls these routines. Collect does not move list cells, but it may "sweep up" and clobber the contents of anything it can't identify as being in use. The arguments to your machine language function are protected, and so is anything placed on the list stack by calling the internal routine push. Everything that is pushed must eventually be popped or LISP/80 will become muddled.

Any LISP built-in function can be called from your function. To discover the addresses of built-in functions, do a GETPROPLIST on the function name and look at the SUBR or FSUBR value.

If you are running CP/M, to load a set of machine language functions into LISP/80, assemble them with the CP/M assembler ASM onto a file called, say, MYFNS.HEX and run LISP with the command "LISP P=MYFNS.HEX".

Under HDOS, use ASM to assemble your functions, creating a file MYFNS.ABS. Then run LISP with the command "LISP P=MYFNS.ABS".

To make your routines available to LISP/80 functions, first compute the entry address of each routine in decimal. Then choose a name for each routine, and use PUTPROP to place on the property list of the routine two properties: N% ARGS, with the number of arguments the routine expects, and SUBR, with the decimal value of the subroutine entry address. The routine may now be called from LISP.

The number of arguments to a machine language function may not exceed 3. If N% ARGS is -1, the routine will be nospread (see Section 4.14). If the subroutine address is placed under the property FSUBR instead of SUBR, it will receive its arguments unevaluated.

Following is a simple machine language function implementing (ADD1 n), which returns n+1. This routine does not check its argument and will return a random value if called with other than a numeric atom. NOTE: the addresses assumed here for ORG and box may differ from the actual values; run "LISP P" to find out the right ones.

```
        ORG 27329       Use the ORG from "LISP P"
ADD1    POP D           Pop return addr.
        POP H           Get argument
        PUSH H          Restore stack
        PUSH D
        INX H           Move to second
        INX H           word (value)
        MOV E,M         and get it in
        INX H           DE.
        MOV D,M
        INX D           Add one to value.
        PUSH D          Push argument to box.
        CALL 19636      Call box to make atom.
        POP B           Pop argument off stack.
        RET             Return the atom in HL.
        END ADD1
```

If this is assembled onto file ADD1.HEX it can be loaded into LISP/80 by the command "LISP P=ADD1.HEX" [under HDOS, use ABS instead of HEX] and linked in by typing to EVALQUOTE

```
PUTPROP (ADD1 SUBR 27329)
PUTPROP (ADD1 N% ARGS 1)
```

5. EDITOR AND FILE PACKAGE

5.1. INTRODUCTION

To make LISP/80 program development easier, a simple program editor and function save routine, written in LISP, are provided. The editor permits editing of function definitions and other s-expressions. The save routine writes the current definitions of a list of functions onto a file, from which they may be reloaded. The system remembers what functions have been loaded from a file, so that the user need not list all the function names when saving them again.

Also provided is PP, a "prettyprint" routine which prints a LISP expression, and in particular a function definition, in a more readable format than is afforded by PRINT.

These functions are written in LISP. They are supplied as files EDIT.LSP and PP.LSP on the LISP/80 distribution disk, and may be loaded by the commands LOAD (EDIT) (or LOAD (B:EDIT) if the file is on B:; under HDOS, LOAD (SY1:EDIT)) and LOAD(PP).

These functions are not particularly sophisticated, fast, or complete. They are provided not only to be used, but also to serve as examples of how LISP can be used to manipulate other LISP programs and to write, in LISP, programs that perform system utility functions. The user may well wish to extend the editor and to polish PP beyond their current state. Or the user may find it easier to make program changes by exiting from LISP/80, editing the program file using PIE or another text editor, and reloading the program.

WARNING: These functions should be loaded early in the LISP session. If memory is almost full loading them may exhaust available storage and the contents of memory may be lost. The amount of storage required by a file may be determined by doing a (COLLECT), loading the file, doing another (COLLECT), and subtracting the new free space counts from the previous ones. Then (COLLECT) may be used to see if the required amount of space is available before loading the file during subsequent LISP runs.

5.2. EDITOR

The functions described in this section must be loaded by LOAD (EDIT) before they can be called.

(EDIT fname). Edits the function definition of fname, using the editing commands shown below. When editing is completed, the function definition is updated to the edited one, and EDIT returns the value fname.

(EDITEXP expr). Similar to EDIT, but edits the actual expression expr. EDITEXP is called by EDIT.

EDIT is an expression editor. It allows inserting, changing and deleting elements of a list. There is always a current expression, which initially is the entire function definition. As editing proceeds, various commands can be used so that one of the sublists of the current expression, or the list containing the current expression, becomes the new current expression. Other commands allow editing the current expression, and it is displayed after every step.

When EDIT is run, it finds the definition of the function fname. If there is no

existing definition, EDIT starts off with the expression (LAMBDA NIL NIL). EDIT prints the current expression, which is the entire function definition, prompts with the character *, and waits for a command.

When the current expression is printed, any list nested at or deeper than the maximum print depth is represented by the character ?. The print depth is initially set to 3 but may be changed by the (P n) command. Example: the expression

```
(LAMBDA (X) (COND ((NULL X) X) (T (CDR X)))))
```

would print as

```
(LAMBDA (X) (COND (? X) (T ?)))
```

This allows complex expressions to be summarized in a reasonable amount of space.

The EDIT commands are:

PP Prettyprints the current expression. This permits viewing the entire expression in a readable format, but is liable to be quite slow. This command requires PP to have been explicitly loaded (see Section 5.4).

n (n is a positive number other than 0). The nth element of the current expression becomes the new current expression. Example: if the current expression is the one shown in the previous paragraph, then the commands

```
3
3
```

would print

```
(COND ((NULL X) X) (T (CDR X)))
(T (CDR X))
```

0 Sets the current expression to the list containing the current expression. Continuing the example of the previous paragraph, typing

```
0
```

would print

```
(COND ((NULL X) X) (T (CDR X)))
```

F Moves forward; i.e., sets the new current expression to be the next list element after the present current expression. If the current expression is C in the list (A B C D E), then the F command moves the current expression to be D. If the current expression is the last element in a list, F prints a ? and does nothing else.

B Moves backward; i.e., sets the new current expression to be the list element preceding the current expression. If the current expression is D in the list (A B C D E), then the B command moves the current expression to be C. If the current expression is the first element in a list, b prints a ? and does nothing else.

(n e1 e2 ... en). The nth element of the current expression is deleted and replaced by the expressions e1 ... en. The replacement is performed using NCONC. Continuing the example, typing

(1 ((ZEROP X) Y) ((NULL X) Z)

prints the edited expression

(COND ((ZEROP X) Y) ((NULL X) Z) (T (CDR X)))

If there are no expressions in the command, the nth element of the current expression is simply deleted. For example, (3) deletes the third element of the current expression.

(-n e1 e2 ... en). The expressions e1 ... en are inserted before the nth element of the current expression, but nothing is deleted. The replacement is performed using NCONC.

(P n) Sets the maximum print depth to n. Expressions nested n levels deep or more are printed as ?.

E Exit from the editor. Note that EDIT alters the list structure of the function definition, so even if EDIT is aborted by typing ctrl-B, any change is likely to be made and irreversible (unless the definition was previously saved on a file or COPYed to another expression.)

5.3. PRETTYPRINT

The functions described in this section must be loaded by LOAD (PP) (or B:PP if the file is on B:) before they can be called.

(PP expr file). Prettyprints the expression expr on file. File is NIL (or omitted) to print on the terminal, or a channel number (see OPENW, Section 4.19) for output to a file or other device.

A prettyprinted expression is considerably more readable than an expression printed by PRINT. However, since it does a considerable amount of character counting, PP is quite slow.

(PPF fname file). Prettyprints the definition of the function fname.

5.4. SAVING FUNCTIONS ON A FILE

The functions described in this section must be loaded by LOAD (EDIT) (or B:EDIT if the file is on B:) before they can be called.

(SAVEFILE fname progs ppflag). Saves function definitions on file fname. The definitions can be read back in by (LOAD fname). The functions which are saved are (1) any functions in the list progs of function names, and (2) any functions which were previously loaded from file fname if fname was previously written by SAVEFILE. The file will contain only these functions and no others; anything previously on the file is lost. If no extension is given for fname, .LSP is assumed.

If ppflag appears and is not NIL, the functions are prettyprinted, using PP.

This will be extremely slow but the resulting file will be more readable. If PP has not explicitly been loaded, PRINT will be used in any event.

When a file created by SAVEFILE is read back in by LOAD, the list of functions defined on the file is remembered (by storing it as the property PROGRAMS on the property list of the file name atom). This allows SAVEFILE to write the functions back out later.

Thus, a function F may be defined originally by typing in a DEFINE. It can be saved the first time by SAVEFILE (MYPROGS (F)), which will create MYPROGS.LSP and write the definition of F to it. Subsequently, the saved definition of F may be restored by LOAD (F), edited, and saved again by SAVEFILE (MYPROGS).

BIBLIOGRAPHY

Laurent Siklossy, Let's Talk LISP. Prentice Hall, Engelwood Cliffs, NJ, 1976. A recommended introduction to LISP.

Daniel Friedman, The Little LISPer. Science Research Associates, 1974. A softcover introduction to LISP using the question and answer method.

Winston, Artificial Intelligence. Addison-Wesley, Reading, MA, 1977. A good hardcover text which describes a number of artificial intelligence applications, with many examples of how to program them in LISP. Few of the programming examples are complete, however, so there are not lots of things for the novice to type in and try to run. The last third of the book contains a good introduction to the LISP language. This is the one book to buy for those learning LISP in order to program AI applications.

Clark Weissman, LISP 1.5 Primer. Dickenson Publishing Co., Belmont, CA, 1967. An old introduction to LISP. It is well written, proceeds slowly, and contains many examples and exercises. All this tends to compensate for its being obsolete in a few places.

John Allen, Anatomy of LISP, McGraw Hill, 1978.

BYTE Magazine, August 1979. Byte Publications, Peterborough, NH. This issue contains a short article introducing LISP, and a number of applications. The introduction is worth reading if the magazine is easily available; but note that it uses FIRST for CAR and REST for CDR. The applications articles may be of interest once the reader has a bit more LISP knowledge.

Warren Teitelman et al, INTERLISP Reference Manual. Xerox PARC, Palo Alto, CA, 1978. Thicker than the Boston telephone book, this manual describes one of the largest LISP systems in existence. It is mainly of academic interest to LISP/80 owners, although if it is available it may be worth looking at, since explanations it provides of the functions LISP/80 does contain are usually applicable to LISP/80 and can be instructive.

INDEX OF FUNCTIONS

This is an alphabetical index to the built-in and library functions described in the LISP/80 Reference Manual. It does not include references to these functions in the introductory sections of the manual.